SECTION 05
Threading

Threading approaches

1. Thread-Runs-Function

2. Thread-Runs-Callable-Object

C++11 supports two different approaches, based on the same API.

Thread-Runs-Function

```
class thread
{
public:
  template<typename Callable>
  explicit thread(Callable func);

  // More...
};
```

The `std::thread` class expects a function address (pointer-to-function) that can be used as the thread function.

The user must supply the function to execute in its own thread of control.

Using Thread-Runs-Function

```
#include <iostream>
#include <thread>

void myThreadFunction()
{
  for(int i = 0; i<100; ++i)
  {
    std::cout << 'x';
  }
}

int main()
{
    std::thread t1(&myThreadFunction);
}
```

Threads are created by supplying the address of the function.

Waiting for a thread to finish

```
class thread
{
public:
  template<typename Callable>
  explicit thread(Callable func);

  // ...

  void join();

  // ...
};
```

We don’t want main to end before any created thread has finished as this leads to undefined behaviour.

We want main to wait until our thread has finished; this is called 'joining'; from the concept for 'fork-and-join'.

`std::thread` supports a `join()` function.

Using join

```
#include <iostream>
#include <thread>

void myThreadFunction()
{
  for(int i = 0; i<100; ++i)
  {
    std::cout << 'x';
  }
}

int main()
{
  std::thread t1(myThreadFunction);
  t1.join();
}
```

`main` will not terminate until `t1` has completed.

Thread with arguments

```
class thread
{
public:
    template<typename Callable>
    explicit thread(Callable func);

    template<typename Callable, typename Args...>
    thread(Callable func, Args... args);

    //…

    void join();

    //…
};
```

A variadic template allows a template function with a variable number of types. Typically the types are deduced from the types of the supplied parameter(s).

Thread with single-argument

```
#include <iostream>
#include <thread>

void myThreadFunction(char c)
{
  for(int i = 0; i<100; ++i)
  {
    std::cout << c;
  }
}

int main()
{
    std::thread t1(myThreadFunction, '1');
    std::thread t2(myThreadFunction, '2');

    t1.join();
    t2.join();
}
```

When the thread object is created the parameters are used to create a `std::function` that holds the thread function. Note, the parameter type supplied must match the parameter type of the thread function (since it is used for template argument deduction).

(For more information on `std::function`, see Appendix L)

Multiple arguments

```
#include <iostream>
#include <thread>

void myThreadFunction(char c, int count)
{
  for(int i = 0; i < count; ++i)
  {
    std::cout << c;
  }
}


int main()
{
    std::thread t1(myThreadFunction, '1', 60);
    std::thread t2(myThreadFunction, '2', 80);

    t1.join();
    t2.join();
}
```

Because of the use of a variadic template on the thread constructor the `std::thread` class can support any number of parameters.

Using a class member function

```
class ThreadedObject
{
public:
  ThreadedObject(char c): ch(c) {}
  void run() const;

private:
  char ch;
};

void ThreadedObject::run() const
{
  for(int i=0; i < 100; ++i)
  {
    std::cout << ch;
  }
}
```

Must pass address of object as first parameter

```
int main()
{
  ThreadedObject obj('.');
  std::thread t1(&ThreadedObject::run, &obj);

  t1.join();
}
```

Similarly, we can use a member function on a class. This is similar to the Thread-Runs-Polymorphic-Object pattern, but we have flexibility in which function is called.

Note, because a `std::function` is (effectively) a generic pointer-to-function we must supply the object's address explicitly as a parameter.

Thread-Runs-Functor

```
#include <iostream>

class ThreadedFunctor
{
public:
    ThreadedFunctor(char c) : ch(c) {}
    void operator()() const;

private:
    char ch;
};

void ThreadedFunctor::operator()() const
{
  for(int i=0; i<100;++i)
  {
    std::cout << ch;
  }
}
```

Remember: overloading `operator()` allows an object to be called as if it were a function (it becomes a *callable object* - see Appendix L for more details)

Using Thread-Runs-Functor

```
#include <thread>

int main()
{
  ThreadedFunctor functorObj1('1');
  ThreadedFunctor functorObj2('2');

  std::thread t1(functorObj1);
  std::thread t2(functorObj2);

  t1.join();
  t2.join();
}
```

This is essentially the same as Thread-Runs-Polymorphic-Object (from a client's perspective) c.f. Appendix H

Thread Functor - alternative

```
#include <thread>

int main()
{
    std::thread t1( (ThreadedFunctor('1')) );
    std::thread t2 = std::thread(ThreadedFunctor('2'));

    t1.join();
    t2.join();
}
```

If you're wondering about the extra parentheses around the `ThreadedFunctor` constructor call, this is to avoid what's known as *C++'s most vexing parse*.

Without the parentheses, the declaration is taken to be a declaration of a function called t1 which takes a pointer-to-a-function-with-no-parameters-returning-an-instance-of-`ThreadedFunctor`, and which returns a `std::thread` object; rather than an object called t1 of type `std::thread` (which is what you would expect)

Functor with parameters

```
#include <iostream>

class FunctorWithParams
{
public:
 FunctorWithParams (char c): ch(c) {}
  void operator()(int count) const;

private:
  char ch;
};

void FunctorWithParams::operator()(int count) const
{
  for(int i=0; i < count; ++i)
  {
    std::cout << ch;
  }
}
```

Since `operator()` is just another member function it can have parameters.

Using functors with parameters

```
#include <thread>

int main()
{
  FunctorWithParams functorObj1('1');
  FunctorWithParams functorObj2('2');

  std::thread t1(functorObj1, 60);
  std::thread t2(functorObj2, 80);
  std::thread t3((FunctorWithParams('3')), 90);

  t1.join();
  t2.join();
}
```

One does have to question why pass parameters to the thread object if you have a functor (with its own state)

Key points

std::thread provides independent units-of execution

A std::thread can be created with any callable object allowing it to model the following threaded patterns

- Thread-Runs-Function
- Thread-Runs-Object

std::thread relies on the client-supplied function or object for its scheduling policy; unlike the Thread-Runs-Polymorphic-Object pattern